Coding Style and Good Computing Practices
Author(s): Jonathan Nagler
Source: *PS: Political Science and Politics*, Vol. 28, No. 3 (Sep., 1995), pp. 488-492
Published by: American Political Science Association
Stable URL: http://www.jstor.org/stable/420315
Accessed: 05-03-2017 01:10 UTC

JSTOR is a not-for-profit service that helps scholars, researchers, and students discover, use, and build upon a wide range of content in a trusted digital archive. We use information technology and tools to increase productivity and facilitate new forms of scholarship. For more information about JSTOR, please contact support@jstor.org.

Your use of the JSTOR archive indicates your acceptance of the Terms & Conditions of Use, available at http://about.jstor.org/terms

*American Political Science Association* is collaborating with JSTOR to digitize, preserve and extend access to *PS: Political Science and Politics*

This content downloaded from 128.59.222.107 on Sun, 05 Mar 2017 01:10:58 UTC
All use subject to http://about.jstor.org/terms

# Coding Style and Good Computing Practices

**Jonathan Nagler,** *University of California, Riverside*

Replication of scholarly analysis depends on individual researchers being able to explain exactly what they have done. And being able to explain exactly what one has done requires keeping good records of it. This article describes basic good computing practices[1] and offers advice for writing clear code that facilitates the task of replicating the research. The goals are simple. First, the researcher should be able to replicate his or her own work six hours later, six months later, and even six years later. Second, others should be able to look at the code and understand what was being done (and preferably why it was being done). In addition, following good computing practices encourages the researcher to maintain a thorough grasp of what is being done with the data. This allows for more efficient research: one is not always rereading one's own work and retracing one's own steps to perform the smallest bit of additional analysis.

This article is *not* meant only for sophisticated statistical researchers. In fact, the statistical procedures you ultimately use have nothing to do with the topic of this article. These practices should be used even if you are doing nothing more complex than producing $2 \times 2$ tables with a particular data set. The sequence this article is written in leads the reader from more general themes to more specific ones. I encourage readers who haven't the patience for the big picture to skip ahead rather than skip the whole article. Even learning basic conventions about variable names will put you ahead of most coders!

First, what do I mean by computing practices and "code?" Computing practices covers everything you do from the time you open the codebook to a data set, or begin to enter your own data, to the time you produce the actual numbers that will be placed in a table of an article. "Code" refers to the actual computer syntax—or computer program—used to perform the computations. This most likely means the set of commands issued to a higher-level statistics package such as SAS or SPSS. A given file of commands is a program. Most political scientists do not think of themselves as computer programmers. But when you write a line of syntax in SAS or SPSS, that is exactly what you are doing—programming a computer. It is coincidental that the language you use is SAS instead of Fortran or C. The paradox is that most political scientists are not trained as computer programmers. And so they never learned basic elements of programming style, nor are they practiced in the art of writing clean, readable, maintainable code. The classic text on programming style remains Kernighan and Plauger, *The Elements of Programming Style* (1974, 1978), and most books on programming include a chapter on style. I recommend including a section on coding style in every graduate methods sequence.

This article starts from the point at which a raw data set exists somewhere on a computer. It breaks analysis into two basic parts: coding the data to put it into a usable form, and computing with the data. It is useful to break the first part into two component steps: reading the essential data from a larger data set; and recoding it and computing new variables to be used in data analysis.

## Lab Books

Our peers in the laboratory sciences maintain lab books. These books indicate what they did to perform their experiments. We are not generally performing experiments. But we have identical goals. We want to be able to retrieve a record of every step we have followed to produce a particular result. This means the lab book should include the name of every file of commands you wrote, with a brief synopsis of what the file was intended to do. The lab book should indicate which produced results worth noting.

It is a good idea to have a template that you follow for each lab book entry; this encourages you to avoid becoming careless in your entries. A template could include date, file, author, purpose, results, and machine. You might have a set of purposes—recoding, data extraction, data analysis—that you feel each file fits into. It may seem superfluous to indicate on what machine the file was executed. But should you develop the habit of computing on several machines, or should you move from one machine to another in the course of a project, this information becomes invaluable in making sure you can locate all your files.

It makes a lot of sense to have the lab book on-line. It can be in either a Wordperfect file, or a plain ascii text file, or whatever you are most comfortable writing in. First, it is easy to search for particular events if you remember their names. Second, it can be accessed by more than one researcher if you are doing joint work.

**Rule:** Maintain a lab book from the beginning of a project to the end.

Lab books should provide an audit trail of any of your results. The lab book should contain all the information necessary to take a given piece of data analysis, and trace back where all of the data came from, including its source and all recode steps. So although you might want to keep many sorts of entries in a lab book, and there are many different styles in which to keep them, the point to keep in mind is that whatever style you choose, it should meet this purpose.



This content downloaded from 128.59.222.107 on Sun, 05 Mar 2017 01:10:58 UTC
All use subject to http://about.jstor.org/terms

## Command Files

The notion of command files may appear as an anachronism to some. After all, can't we all just work interactively by pointing and clicking to do our analysis, not having to go through the tedium of typing separate lines for each regression we want to run and submitting our job for analysis? Yes, we can. But, we probably don't want to. And even if we do, modern software will keep a log for us of what we have done (a fact that seems to escape many users of the software). The reason I am not a fan of interactive work has to do with the nature of the analysis we do. The model we ultimately settle on was usually arrived at after several estimates testing many models. And this is appropriate: we all ought to be testing the robustness of our results to changes in model specification, changes in variable measurement, and so forth. By writing command files to perform estimates, *and keeping each version*, one has a record of all this.

You generally want a large set of comments at the beginning of each file, indicating what the file is intended to do. And each file should not do too much. Comments on the top of a file should list the following:

1. State the date it was written, and by whom.
2. Include a description of what the file does.
3. Note from what file it was immediately derived.
4. Note all changes from its predecessors, where appropriate.
5. Indicate any data sets the file uses as input. If the file uses a data set, there should be a comment indicating the source of the data set. This could be either another file you have that produced the data or a description of a public-use data set. If it is a public-use data set, include the date and version number.
6. Indicate any output files or data set created.

For example, the first nine lines of a command file might be

```
\* File-Name:    mnp152.g
   Data:         Feb 2, 1994
   Author:       JN
   Purpose:      This file does multinomial
                 probits on our basic model.
   Data Used:    nes921.dat (created by
                 mkascic.cmd)
   Output File:  mnp152.out
   Data Output:  None
   Machine:      billandal (IBM/RS6000)
\*
```

You could keep a template with the fields above left blank, and read in the template to start each new command file. You should treat the comments at the top of a file the way you would the notes on a table; they should allow the file to stand alone and be interpreted by anyone opening the file without access to other files.

## Know the Goal of the Code

It makes no sense to start coding variables if you do not know what the point of the analysis using the variables will be. You end up with a bunch of variables that are all being recoded later on and confusing you to no end. Before you start manipulating the data, figure out what you will be testing and how the variables need to be set up.

Example: say you want to test the claim that people who voted in 1992, but did not vote in 1988, were more likely to support Perot than Bush in 1992. How do you code the independent variable indicated here? Well, you really want a "new voter" variable, because your substantive hypothesis is stated most directly in terms of "new voters." Thus, the variable should be coded so that

vote in 1992, not voted in 1988 = 1
voted in 1992, voted in 1988 = 0

> **Rule:** Code each variable so that it corresponds as closely as possible to a verbal description of the substantive hypothesis the variable will be used to test.

## Fixing Mistakes

If you find errors, the errors should be corrected where they first appear in your code—not patched over later. This may mean rerunning many files. But this is preferable to the alternative. If you patch the error further downstream in the code then you will need to remember to repeat the patch should you make any change in the early part of the data preparation (i.e., you decide you need to pull additional variables from a data set, etc.) If the patch is downstream, you are also likely to get confused as to which of your analysis runs are based on legitimate (patched) data, and which are based on incorrect data.

> **Rule:** Errors in code should be corrected where they occur and the code rerun.

## Data Manipulation Versus Data Analysis

Most data go through many manipulations from raw form to the form we use in our analyses. It makes sense to isolate as much of this as possible in a separate file. For instance, suppose you are using the 1992 National Election Study (NES) presidential election file. You have 100 variables in mind that you *might* use in your analysis. It makes sense to have a program that will pull those 100 variables from the NES data set, give them the names you want, do any basic recoding that you know you will maintain for all of your analysis, and create a "system file" of these 100 named and recoded variables that can be read by your statistics package. There are at least two reasons for this. First, it saves a lot of time. You are going to estimate at least 50 models before settling on one. Do you really want to read the whole NES data set off the disk 50 times when you could read a file 1/20th the size instead? Second, why do all that recoding and naming 50 times? You might accidentally alter the recodes in one of your files.

> **Rule:** Separate tasks related to data manipulation vs. data analysis into separate files.

## Modularity

Separating data manipulation and data analysis is an example of mod-



This content downloaded from 128.59.222.107 on Sun, 05 Mar 2017 01:10:58 UTC
All use subject to http://about.jstor.org/terms

ularity. Modularity refers to the concept that tasks should be split up. If two tasks can be performed sequentially, rather than two at a time, then perform them sequentially. The logic for this is simple. Lots of things can go wrong. You want to be able to isolate what went wrong. You also want to be able to isolate what went right. After what specific change did things improve? Also, this makes for much more readable code.

Thus if you will be engaging in producing some tables before multivariate analysis, you might have a series of programs: descrip1.cmd, descript2.cmd, . . . , descrip9.cmd. Following this, you might produce: reg1.cmd, reg2.cmd, . . . , reg99.cmd. You need not constrain yourself to one regression per file. But the regressions in each file should constitute a coherent set. For instance, one file might contain your three most likely models of vote choice, each disaggregated by sex. This does tend to lead to proliferation of files. One can start with reg1.cmd and finish with reg243.cmd. But disk space is cheap these days, and the files can easily be compressed and stored on floppies if disk space is getting tight.

**Rule:** Each program should perform only one task.

## KISS

Keep it simple and don't get too clever. You may think of a very clever way to code something this week. Unfortunately you may not be as clever next week and you might not be able to figure out what you did. Also, the next person to read your code might not be so clever. Finally, you might not be as clever as you think—the clever way you think of to do three steps at once might only work for five out of six possible cases you are implementing it on. Or it might create nonsense values out of what should be missing data. Why take the chance? Computers are very fast. Any gains you make through efficiency will be dwarfed by the confusion caused later on in trying to figure out what exactly your code is doing so efficiently.

**Rule:** Do not try to be as clever as possible when coding. Try to write code that is as simple as possible.

## Variable Names

There is basically no place for variables named **X1** other than in simulated data. Our data are real; they should have names that impart as much meaning as possible. Unfortunately many statistical packages still limit us to eight-character names (and for portability's sake, we are forced to stick with eight-character names even in packages that don't impose the limit). However, your keyboard has 84 keys, and the alphabet has 52 letters: 26 lower-case and 26 upper-case. Indulge yourself and make liberal use of them. There are also several additional useful characters—such as the underscore and the digits 0–9—at your disposal. It is a convention in programming to use UPPER-CASE characters to indicate constants and lower-case characters to indicate variables. This might not be as useful in statistical programming. You might adopt the convention that capitals refer to computed quantities (such as **PROBCHC1:** the estimated PROBability of CHoosing Choice 1). And if you are trying to have your code closely follow the notation of a particular econometrics article, you might use a capital U for utility, or a capital V for the systemic component of utility. Obviously in such a case comments would be in order! Some people like variable names such as **NatlEcR** because the use of capitals allows for clearly indicating where one word stops and another starts. **NatlEcR** makes it easier to think of 'National Economic—Retrospective' than **natlecr** might. You will need to make some tradeoffs in the conventions you choose. The important thing is to adopt a convention on the use of capitals and stick with it.

**Rule:** Use a consistent style regarding lower- and upper-case letters.

**Rule:** Use variable names that have substantive meaning.

When possible a variable name should reveal subject and direction. The simplest case is probably a dummy variable for a respondent's gender; imagine it is coded so that 0 = men, 1 = women. We could call the variable either "SEX," or "WOMEN." It is clear that "WOMEN" is the better name because it indicates the direction of the variable. When we see our coefficients in the output we won't have to guess whether we coded men = 1 or women = 1.

**Rule:** Use variable names that indicate direction where possible.

Similarly, value labels are useful for packages that permit them. The examples of computer syntax I use in this article are written in SST (Dubin/Rivers 1992), but they can be translated easily into SAS, SPSS, or most statistical packages. Here is a simple example. The variable **natlecr** indicates the respondent's retrospective view of performance of the national economy. Notice that the variable name can indicate only so much information in 8 characters. But the label of it and the values tell us what we need. And the fact that the label tells us where to look the variable up in the code book is further protection.

```
label   var[natlecr]\
   lab[v3531:national economy – retro]\
   val[1 gotbet 3 same 5 gotworse]\
```

Some people using NES data—or any data produced by someone else accompanied by a codebook—follow the convention of naming the variable by its codebook number (i.e., V3531), and using labels for substantive meaning. I think this is a poor practice. Consider which of the following statements is easier to read:

```
logit   dep[preschc]\
            ind[one educ women partyid]
```

or:

```
logit   dep[V5609]\
            ind[one V3908 V4201 V3634]
```

The codebook name for the variable should *definitely* be retained; but it can be retained in the label statement. Without the codebook name one would not know which of the several party-identification variables the variable **partyid** refers to.



This content downloaded from 128.59.222.107 on Sun, 05 Mar 2017 01:10:58 UTC
All use subject to http://about.jstor.org/terms

## Writing Cleanly

Anything that reduces ambiguity is good. Parentheses do so and so parentheses are good. A reader should not need to remember the precedence of operators. But in most cases parentheses are more valuable as visual cues to grouping expressions than to actual precedence of operators. Almost as useful as parentheses is white space. Proper spacing in your code can make it much easier to read. This means both vertical space between sections of a program that do different tasks, and indenting to make things easier to follow.

> **Rule:** Use appropriate white space in your programs, and do so in a consistent fashion to make them easy to read.

## Comments

There is probably *nothing* more important than having adequate comments in your code. Basically one should have comments before any distinct task that the code is about to perform. Beyond this, one can have a comment to describe what a single line does. The basic rule of thumb is this: is the line of code absolutely, positively self-explanatory *to someone other than yourself* without the comment? If there is any ambiguity, go ahead and put the comment in.

Remember though, the comments should *add* to the clarity of the code. Don't put a comment before each line repeating the content of the line. Put comments in before specific blocks of code. Only add a comment for a line where the individual line might not be clear. And remember, if the individual line is not clear without a comment, maybe you should rewrite it.

> **Rule:** Include comments before each block of code describing the purpose of the code.

> **Rule:** Include comments for any line of code if the meaning of the line will not be unambiguous to someone other than yourself.

> **Rule:** Rewrite any code that is not clear.

Following is a case where a single comment lets us know what is going on. Most programmers think that well-written code should be self-documenting. This is partly true. But no matter how well written your code is, some comments can make it much clearer.

```
rem   ********************************************************
rem   Create party-id dummy variables
rem   ********************************************************
rem   Missing values are handled correctly here by SST.
rem   In other statistics packages these three variables might
rem   have to be initialized as missing first.
set   dem = (pid < 3)
set   ind = (pid == 3)
set   rep = (pid > 3)
rem   ********************************************************
rem   ********************************************************
```

## Recodes and Creating New Variables

Probably the most important thing to keep track of both when recoding variables and creating new variables is missing data. There is no general rule that can specify exactly how to do this, because treatment of missing data can vary across statistics packages. Thus the best rule is:

> **Rule:** Verify that missing data are handled correctly on any recode or creation of a new variable.

In some statistics packages you may be best served by initializing all new variables as missing data, and allowing them to become legitimate values only when they are assigned a legitimate value. The best advice is to recode and create new variables defensively.

> **Rule:** After creating each new variable or recoding any variable, produce frequencies or descriptive statistics of the new variable and examine them to be sure that you achieved what you intended.

Generally it is poor style to hardwire values into your code. Any specific values are likely to change when some related piece of code somewhere else is altered or when the data set changes.

> **Rule:** When possible, automate things and avoid placing hard-wired values (those computed "by hand") in code.

Finally, after you have done recodes and created new variables it is a good idea to list all the variables. This way, you can confirm that you and your statistics package agree on what data are available and how many observations are available for each variable. In SST this would be done with a list command; in SAS, PROC CONTENTS will produce a clean list of variables. Most statistics packages offer similar commands.

## Procedures or Macros

Most political scientists do not ever have to write a macro or procedure, but maybe that's why they do so little secondary analysis once they generate some estimates. The purpose of a well-defined procedure is to automate a particular sequence of steps. Procedures and macros are useful both in making your code more readable and in allowing you to perform the same operation multiple times on different values or on different variables. The use of procedures and macros is a topic for a separate article, but political scientists should realize that the tools are available in most statistics packages.

## Summing Up

The rules presented here represent *one way* of accomplishing the goal you should have in mind. That goal is to write clear code that will function reliably and that can be read and understood by you and others and can serve as a road map



This content downloaded from 128.59.222.107 on Sun, 05 Mar 2017 01:10:58 UTC
All use subject to http://about.jstor.org/terms

for replicating and extending your research.

Most people are in a huge hurry when they write their code. Either they are excited about getting the results and want them as fast as possible, or they figure the code will be run once and then thrown out. **If your program is not worth documenting, it probably is not worth running.** The time you save by writing clean code and commenting it carefully may be your own.

## Rules

1. Maintain a lab book from the beginning of a project to the end.
2. Code each variable so that it corresponds as closely as possible to a verbal description of the substantive hypothesis the variable will be used to test.
3. Correct errors in code where they occur, and rerun the code.
4. Separate tasks related to data manipulation vs. data analysis into separate files.
5. Design each program to perform only one task.
6. Do not try to be as clever as possible when coding. Try to write code that is as simple as possible.
7. Set up each section of a program to perform only one task.
8. Use a consistent style regarding lower- and upper-case letters.
9. Use variable names that have substantive meaning.
10. Use variable names that indicate direction where possible.
11. Use appropriate white space in your programs, and do so in a consistent fashion to make the programs easy to read.
12. Include comments before each block of code describing the purpose of the code.
13. Include comments for any line of code if the meaning of the line will not be unambiguous to someone other than yourself.
14. Rewrite any code that is not clear.
15. Verify that missing data are handled correctly on any recode or creation of a new variable.
16. After creating each new variable or recoding any variable, produce frequencies or descriptive statistics of the new variable and examine them to be sure that you achieved what you intended.
17. When possible, automate things and avoid placing hard-wired values (those computed "by hand") in code.

## Note

1. A version of this article appeared in *The Political Methodologist*, vol. 6, no. 2, spring 1995, 2–8.

I thank Charles Franklin, Bob Hanneman, Gary King, and Burt Kritzer for useful comments and suggestions.

## References

Dubin, Jeffrey, and R. Douglas Rivers. 1992. *Statistical Software Tools Users Guide: Volume 2.0.* Pasadena, CA: Dubin/Rivers Research.

Kernighan, Brian W. and P. J. Plauger. 1978. *The Elements of Programming Style, 2nd edition*. New York: McGraw-Hill.

## About the Author

**Jonathan Nagler** is associate professor at University of California, Riverside. His research interests include qualitative analysis, campaigns and elections. He can be reached at nagler@wizard.ucr.edu.

# Response: Potential Research Policies for Political Science

**Paul S. Herrnson,** *University of Maryland, College Park*

As evidenced by the thoughtful symposium participants, political scientists have a variety of opinions about verification, replication, and data archiving. I continue to have strong reservations regarding the proposed replication, verification, data relinquishment rules debated here. The symposium, however, has convinced me of one important thing: a still broader discussion is needed before editors or organizations change the norms governing research or publication. Walter Stone's (1995) proposal to survey political scientists to obtain their opinions on verification/replication is, therefore, a good one.

The survey should ask political scientists to consider a variety of policies applicable to quantitative studies, including the following:

1. *A "True" Replication Policy*
   Journals would reserve a certain amount of space for studies that replicate a piece of research in its entirety, including data collection and analysis. Authors who seek to publish studies that are based on original data would be required to provide more details on their data collection process than are included in articles currently published in most political science journals, making it easier for others to replicate the original research. Neulip (1991) is an example of a publication based entirely on replication.
2. *A Verification Policy*
   Journals would require scholars to include with their manuscript submissions the printout from which their statistical results were generated. The printout would include variable distributions, scatterplots, and other



This content downloaded from 128.59.222.107 on Sun, 05 Mar 2017 01:10:58 UTC
All use subject to http://about.jstor.org/terms